Parveen Khurana

124 Followers About Following

CNN Architectures — Part 2

Parveen Khurana Feb 17, 2020 · 11 min read

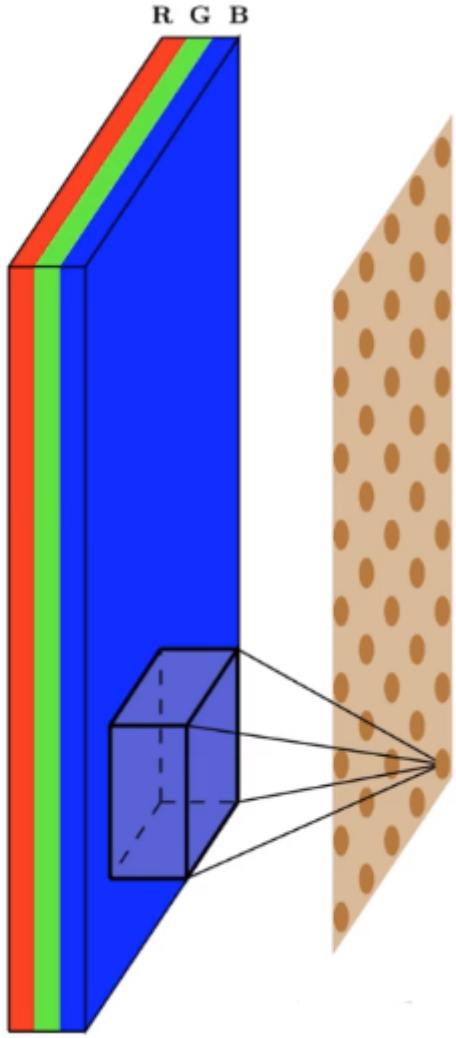
This article covers the content discussed in the CNN Architectures(Part 2) module of the Deep Learning course and all the images are taken from the same module.

Setting the Context:

As discussed in the previous article, we use the filter of size '**3**' in **VGG-16** or **VGG-19**, **one question** that arises at this point is **why to use filters of size '3', why not of size '5' or say '7'**. This is one of the questions that we will discuss in this article. Another issue that we have is that the number of parameters in the first fully connected layer in the VGG network is near about 102M, so we will discuss how to reduce the number of parameters in this layer. And this thing will also be helpful even if we have a more number of layers in the network, and if we are able to reduce the number of parameters in the first fully connected layer, then we can also think about using the filter of size '**5**' and '**7**'(which otherwise would have resulted in an increase in the number of parameters) in addition to the filters of size '**3**'. We will also discuss how to compute the number of computations in this article. So, our main three points to discuss in this article are:

- Number of parameters
- Number of computations
- Number of layers to have in the network
- Filter size to be used

we use a stride of **'1'** and use appropriate padding so that the output dimensions are the same as the input dimensions.



The application of one filter will give us one 2D feature map. To compute one pixel in this feature map, we need to do '**F X F X D**' computations; therefore, for the entire output(of the same dimensions as the input), we need to do '$\mathbf{W_I}$ **X** $\mathbf{H_I}$ **X F X F X D**' computations.

Assume $S = 1$ and we have used appropriate padding so that $W_O = W_I = W$ and $H_O = H_I = H$

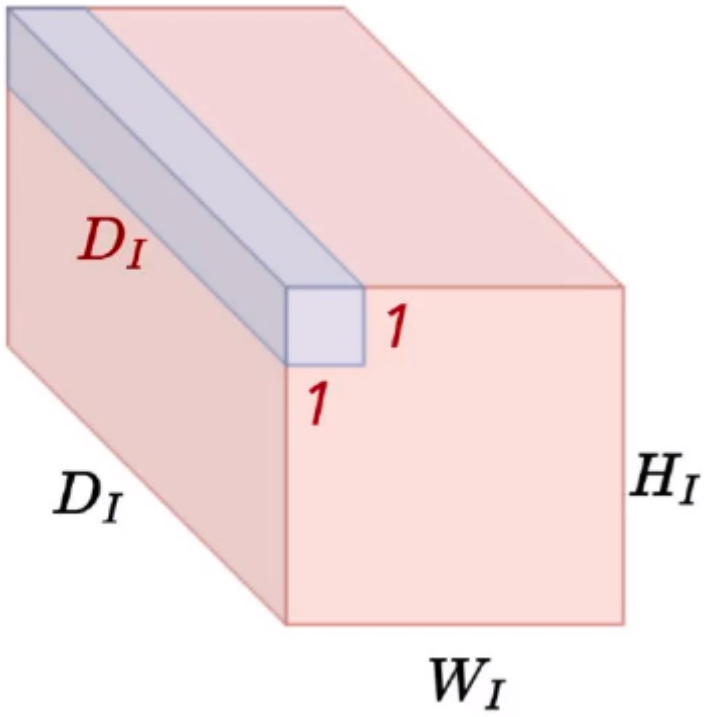
the depth, we should also control **'W, H'** but they depend on the original input and the number of max-pooling layers we use but **depth depends on number of filters** that we use and **if we use more filters, the number of computations would be high, number of parameters would be high.**

1 × 1 Convolutions:

The problem that we are facing is that if the depth of the input volume is very high and if we want to apply convolutional operation on top of that, then the number of computations would be very high because this convolution will operate along the depth and we want to see if somehow we can reduce the depth.

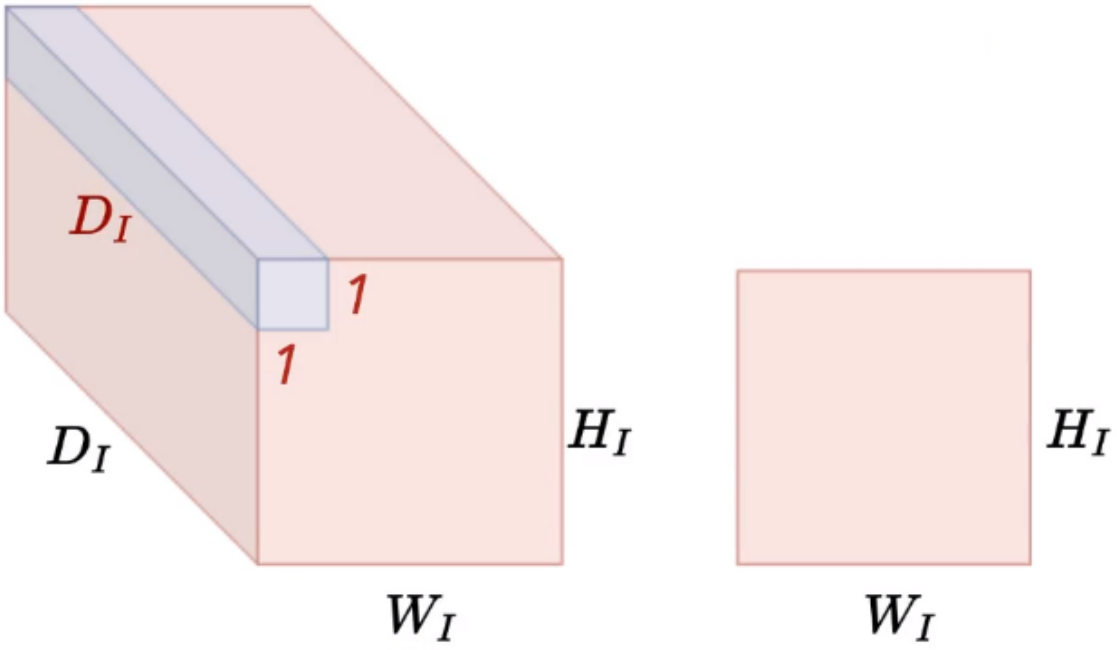
Why would the depth be high to start with?

If we are using multiple filters of different sizes say we are using 100 '**3 X 3**' filters at that layer, 100 '**5 X 5**' filters at that layer, 100 '**7 X 7**' filters and so on, the number of filters would be very high and the depth of the output layer would increase and this output volume would then act as the input volume for the next layer and eventually, the number of computations would be high. This is where '**1 X 1**' convolution is used.



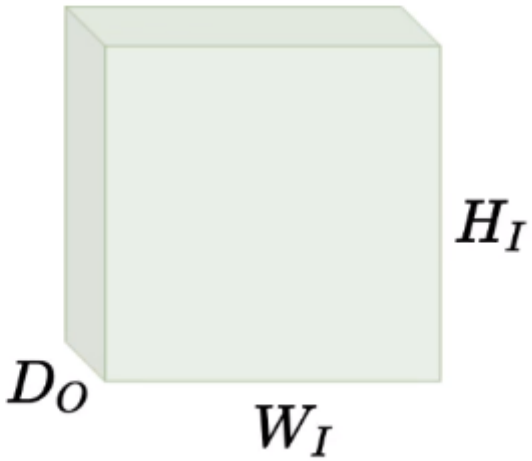
'**1 X 1**' kernel just takes the average of '**1 X 1**' neighborhood which is just the pixel itself. So, what this **kernel actually does is that it averages across the depth**. It computes

input volume.



In a way, the depth of the input volume has been compressed, it just becomes a 2D output instead of a 3D output.

If we use $D_O (< D_I)$ such filters we will get output volume of size $W_I \times H_I \times D_O$

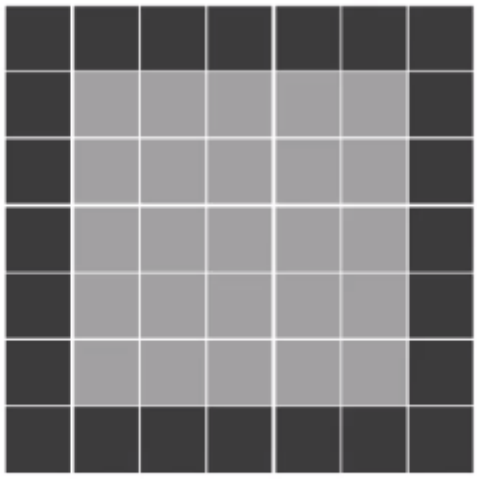


we have shrunk this input volume of large depth to a volume having small depth while in some sense retaining all the information of the input volume as we have just taken a weighted average of all the pixels along the depth.

Now if this new volume is used as the input volume for the next layer, then the number of computations is going to be low compared to earlier as we have reduced the depth.

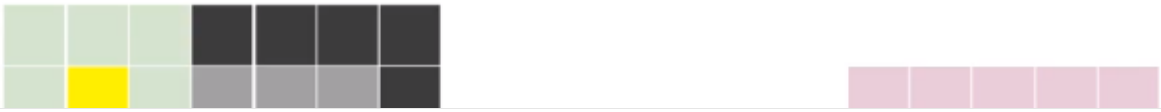
In effect, we can take advantage of the situation by first taking the different representation of the input volume by using different filters of different size and then we can just take the weighted average along the depth to compress the output volume which would act as the input volume for the next layer.

Till now, we have seen the max pooling operation with a stride of **2** which effectively reduces the dimensions of the input volume by half. We could also perform this operation using a stride of **1**. Let's say the input volume is of dimensions '**5 X 5**' and we are using padding of **1** (black pixels in below image) and filter size of **3**.



Input

If we perform max-pooling with a stride of **1** and appropriate padding, we can be sure that the resulting volume is going to of the same size as the input volume as depicted below:



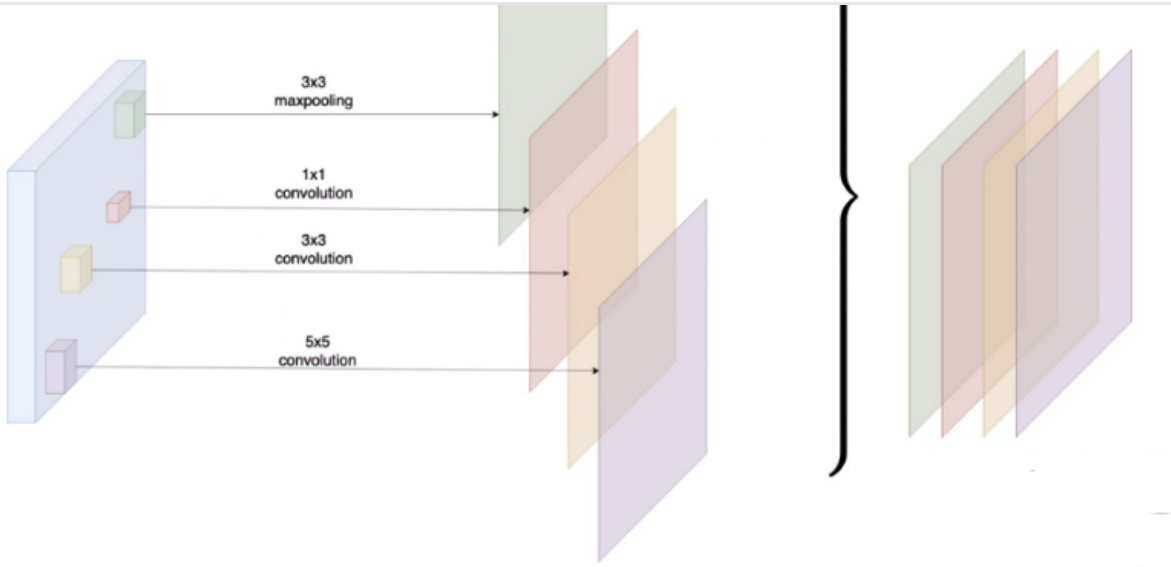
The intuition behind GoogleNet:

In most of the standard network architectures that we have, the intuition is not clear why and when to perform the max-pooling operation, when to use the convolutional operation. For example, in **AlextNet** we have the convolutional operation and max-pooling operation back to back whereas, in **VGGNet**, we have 3 convolutional operations back to back then 1 max-pooling layer.

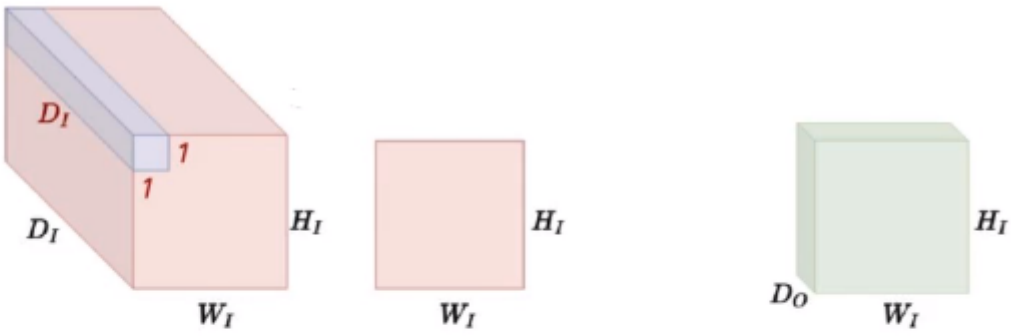
What GoogleNet does is that it performs all these operations at the same time meaning for a given input volume, we compute the max-pooling output, convolutional output using different filter sizes(we don't choose which one to use, we just apply all of them at the same time), say we are using D_1 '**1 X 1**' filters, D_2 '**3 X 3**' filters, D_3 '**5 X 5**' filters and we use a Stride of **1** and appropriate padding so that the size of the output volume is same as the size of the input for all of the size of the filters (s) and max-pooling layers.

We would be getting one feature map from max-pooling operation on the input, $\mathbf{D_1}$ features maps from '**1 X 1**' filters, $\mathbf{D_2}$ feature maps from '**3 X 3**' filters, $\mathbf{D_3}$ feature maps from '**5 X 5**' filters and if we stack these feature maps into one output volume, we have the depth of this output volume as $(\mathbf{1 + D_1 + D_2 + D_3})$. So, we have captured information in this output volume using multiple filters of different sizes and from max-pooling operation as well. So, instead of choosing what layer to use, what filter size to use at a particular layer, **the idea behind GoogleNet is to use all the operations at the same time**.

One question that arises here is why not to use '**7 X 7**' filter or '**11 X 11**' filter, one conclusion that VGG Net gave us was that if we have many back to back convolutional layers then we don't need a larger filter in the initial layers, we could have small filters. That's why we don't have larger filters in GoogleNet and we are applying all the other filters at the same time.



The depth of the output volume, in this case, is (**1 + D1 + D2 + D3**). If the values of **D1**, **D2**, **D3** are large, then we have a problem here as this volume act as the input for the next layer and if the depth of this layer is large then the number of computations in the next layer is going to be large and to deal with this, we would use '**1 X 1**' kernels.

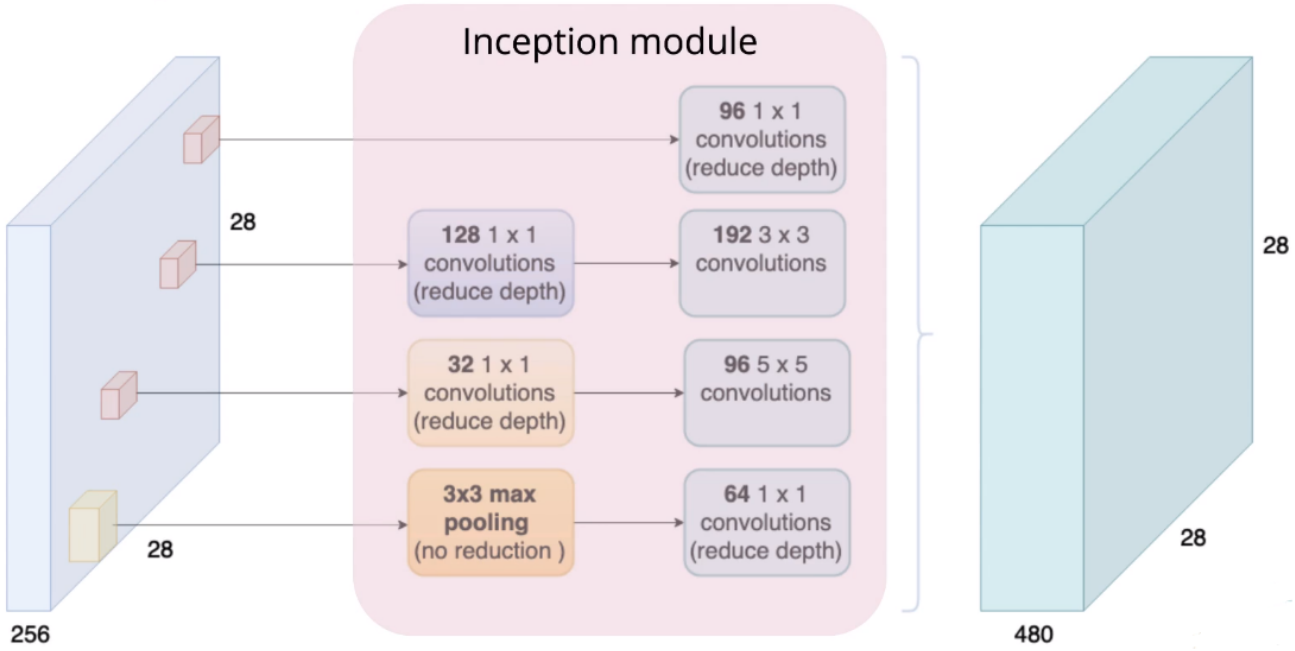


Use 1 x 1 convolutions to reduce the depth and hence the number of computations

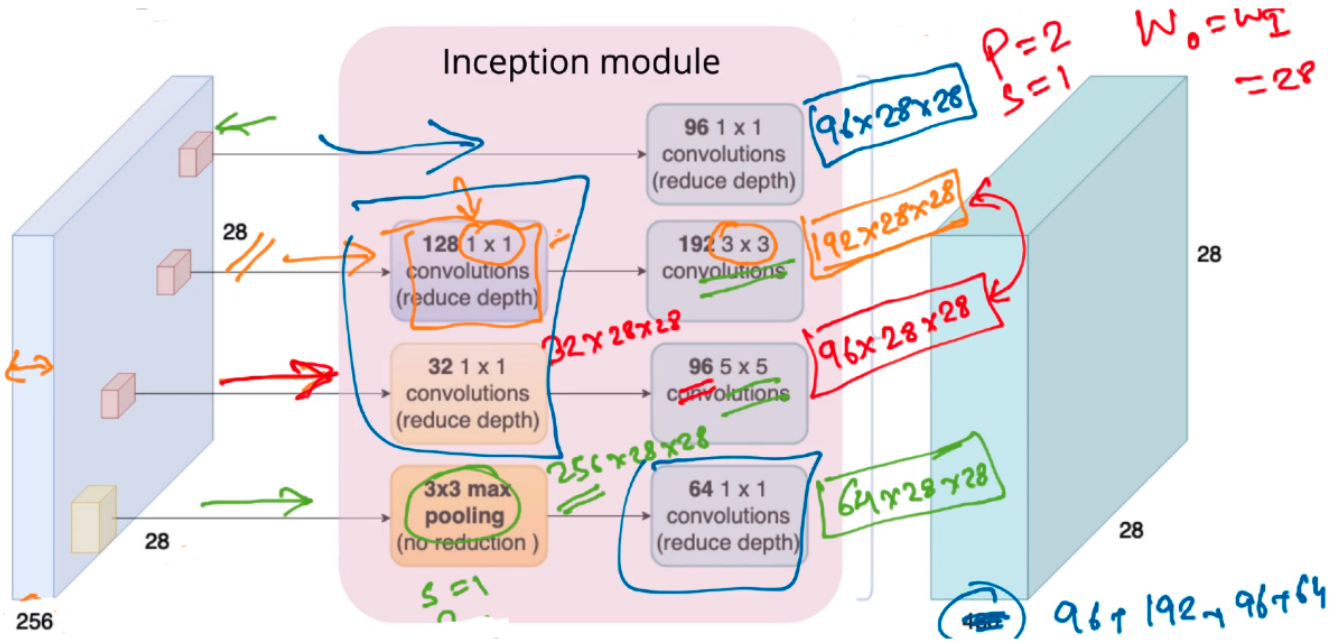
So, after applying these '**1 X 1**' filters, we then apply multiple convolutional layers, max-pooling on this output volume which acts as the input for the next layer.

Inception Module:

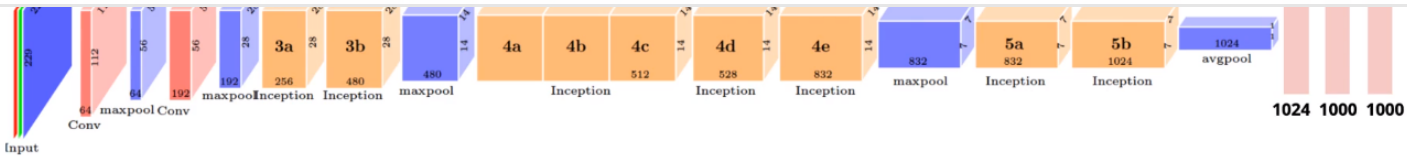
Let's say at some point, we have the dimensions of the input volume as '**28 X 28 X 256**', we will apply the operation depicted below to this input volume all at the same time in such a way that the output width and height are the same at all the points in the network after any of the operations.



The dimension of the output volume here would be 448 instead of 480.



GoogleNet Architecture:

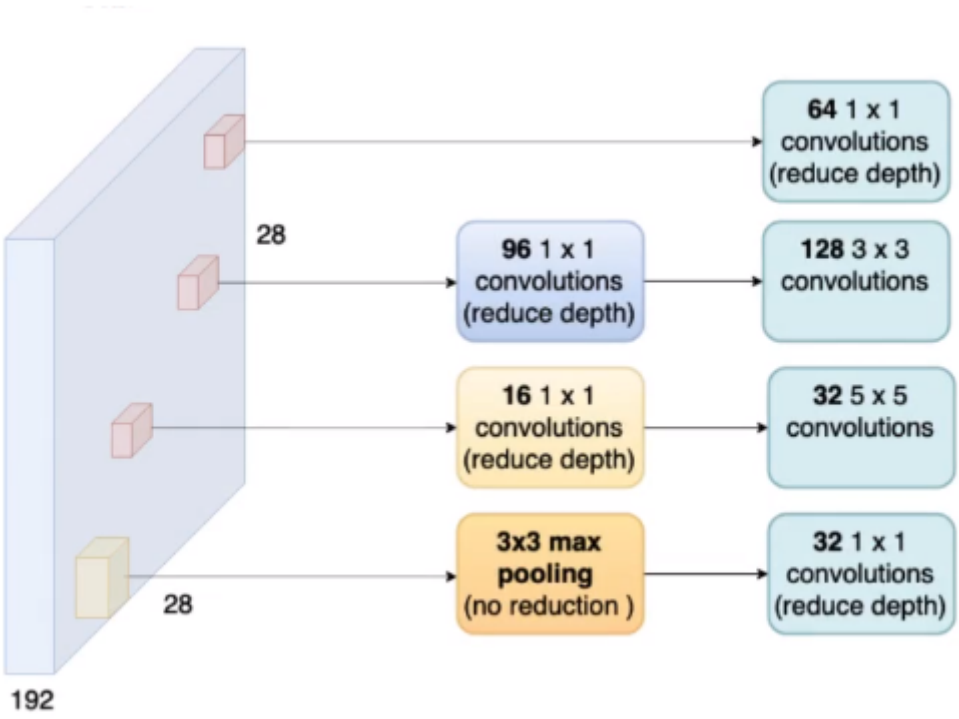


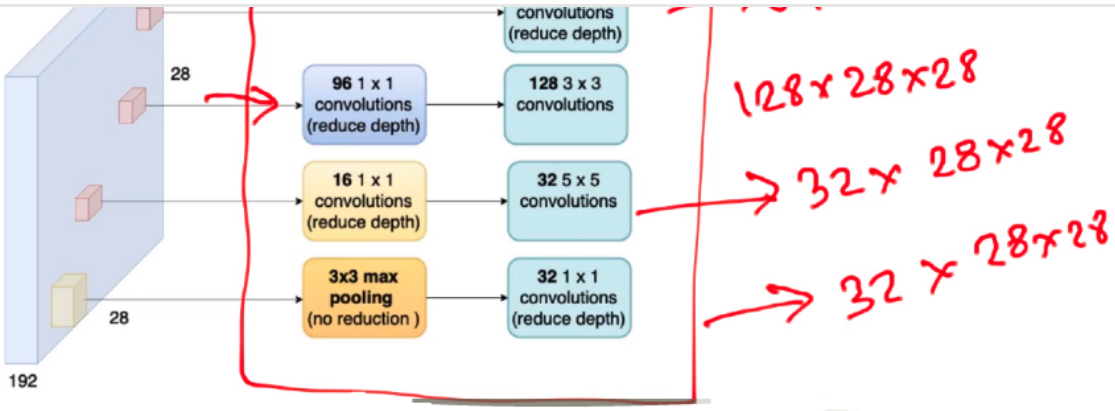
Input dimensions are '224 × 224 × 3'.

Operations on the input are performed as shown in the above image. First, we apply the convolutional operation on the input, the number of filters used are 64 with a filter size of '**3**' and stride of '**2**' which reduces the size of the input by half. Then max pooling is used with appropriate values of stride and padding so that we get the dimensions as depicted in the above image.

The Inception block shown in the above image passes the input volume to that block through the inception module performing all the operations as discussed in the above paragraph of the Inception module.

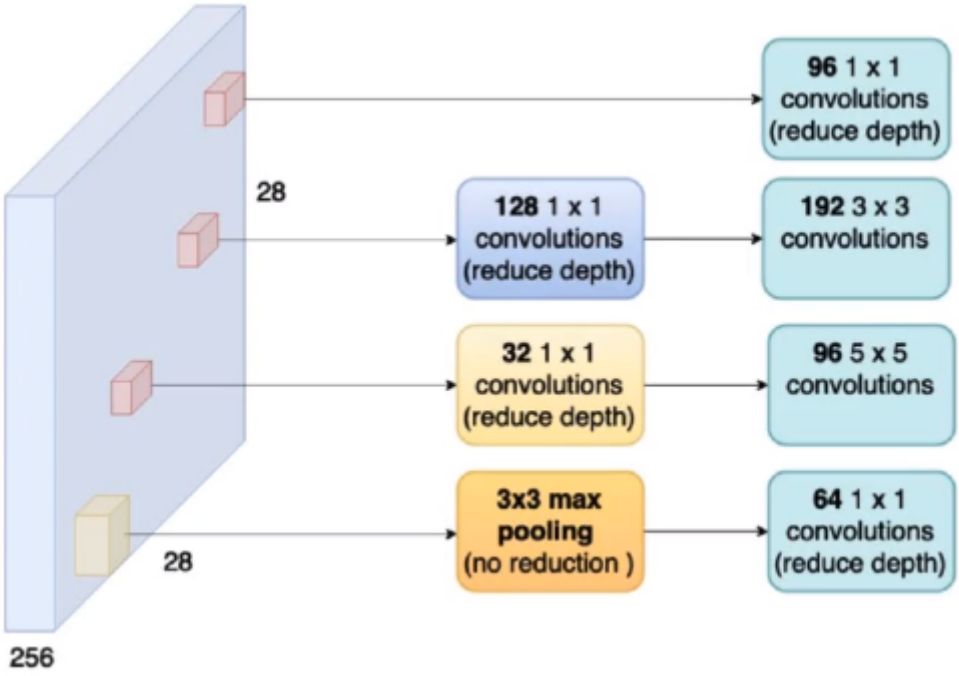
Inception module **3(a)** in the above figure looks like the below:





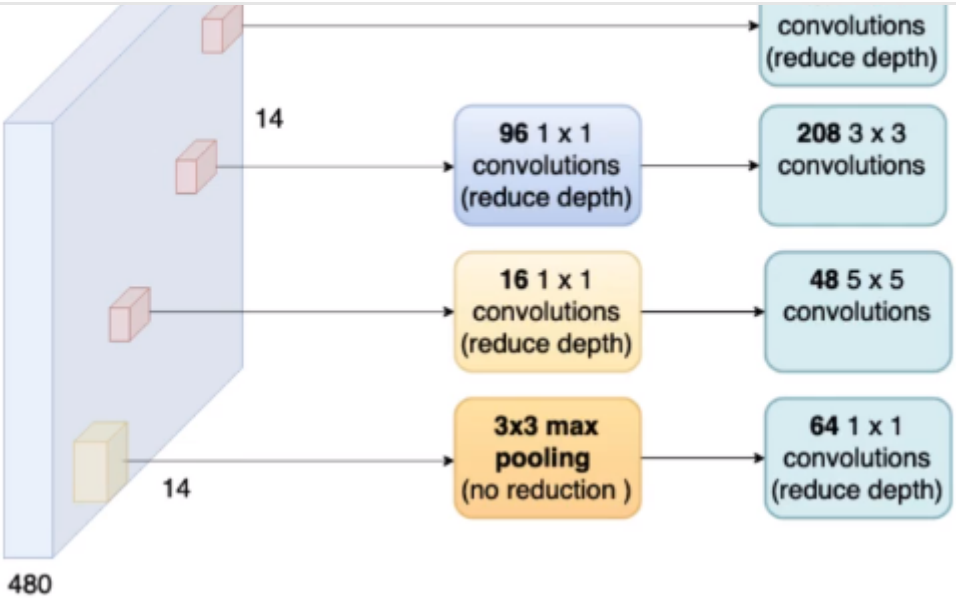
Across different inception modules in the google net architecture, the only thing going to change is the number of filters in each module, the overall blueprint of using the '**1 X 1**', '**3 X 3**', '**5 X 5**' kernels remains the same along with the '**3 X 3**' kernel for max-pooling.

Inception Module **3(b)**



Max pooling inside the inception module does not help with the reduction in the width and the height because stride being used there is 1, so we use max-pooling outside inception module also which helps to reduce the dimensions.

Inception Module **4(a)**

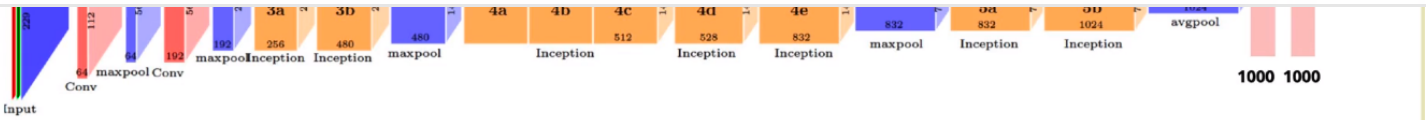


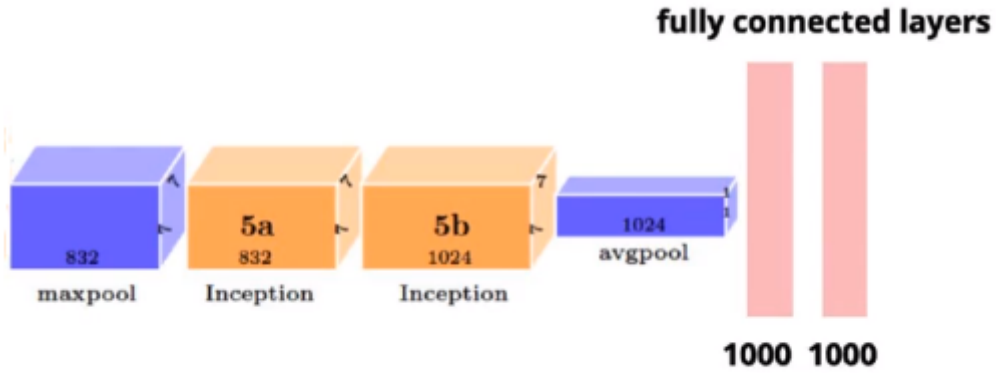
Similarly, we pass the input through all of the layers and after the last inception module i.e after **5(b)**, we have the dimensions as '**1024 X 7 X 7**', now in case of VGG Net we had this final dimension as '**512 X 7 X 7**' and it was fully connected with a '4096' dimensional vector and that's where we had the maximum number of parameters. To avoid the same scenario here in GoogleNet, we use Average pooling.

The way GoogleNet deal with this large number of parameters in the first fully connected layer is that we consider the final dimensions of the output volume as we have **1024** channels each of size '**7 X 7**' or each channel having 49 pixels, GoogleNet takes the average of all these 49 pixels that we have in each of the channels and therefore we are left with just that single value(average of all the pixels) in each of the channels and overall we would have this average value across all of the **1024** channels or in other words, after the average pooling layer, we would be left with '**1024 X 1 X 1**' output volume.

Now if we connect this output volume using a fully connected layer to a '**1000**' dimensional vector, then the number of neurons for this fully connected layer would be '**1024 X 1000**' which is near to 1M as opposed to 49M what would have been if we don't use average pooling and connect the output volume as it is with the next layer.

After this, one more layer of **1000** neurons are there, on which we apply the softmax to get the final output.





Few properties of GoogleNet:

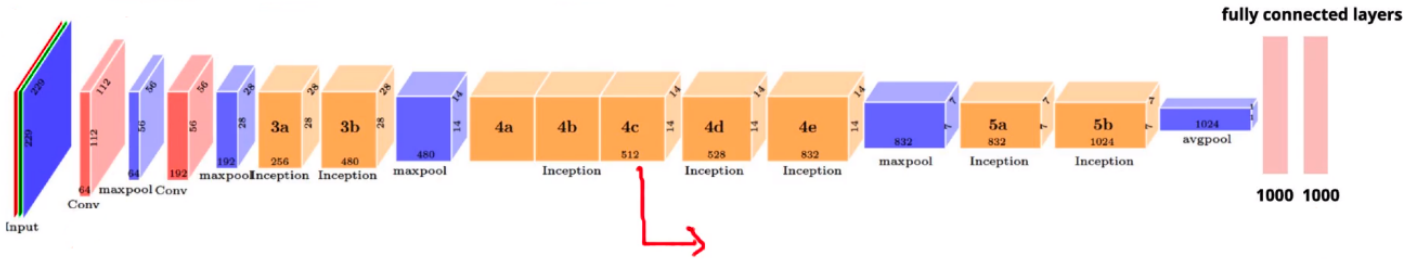
- It uses multiple convolutions and max-pooling in parallel so the question of the choice of filters is taken care of.
- To reduce the number of parameters, it uses Average pooling.
- '1 X 1' convolutions are used to reduce the number of computations.

Total number of layers in GoogleNet: 22

Auxiliary Loss for training a deep network:

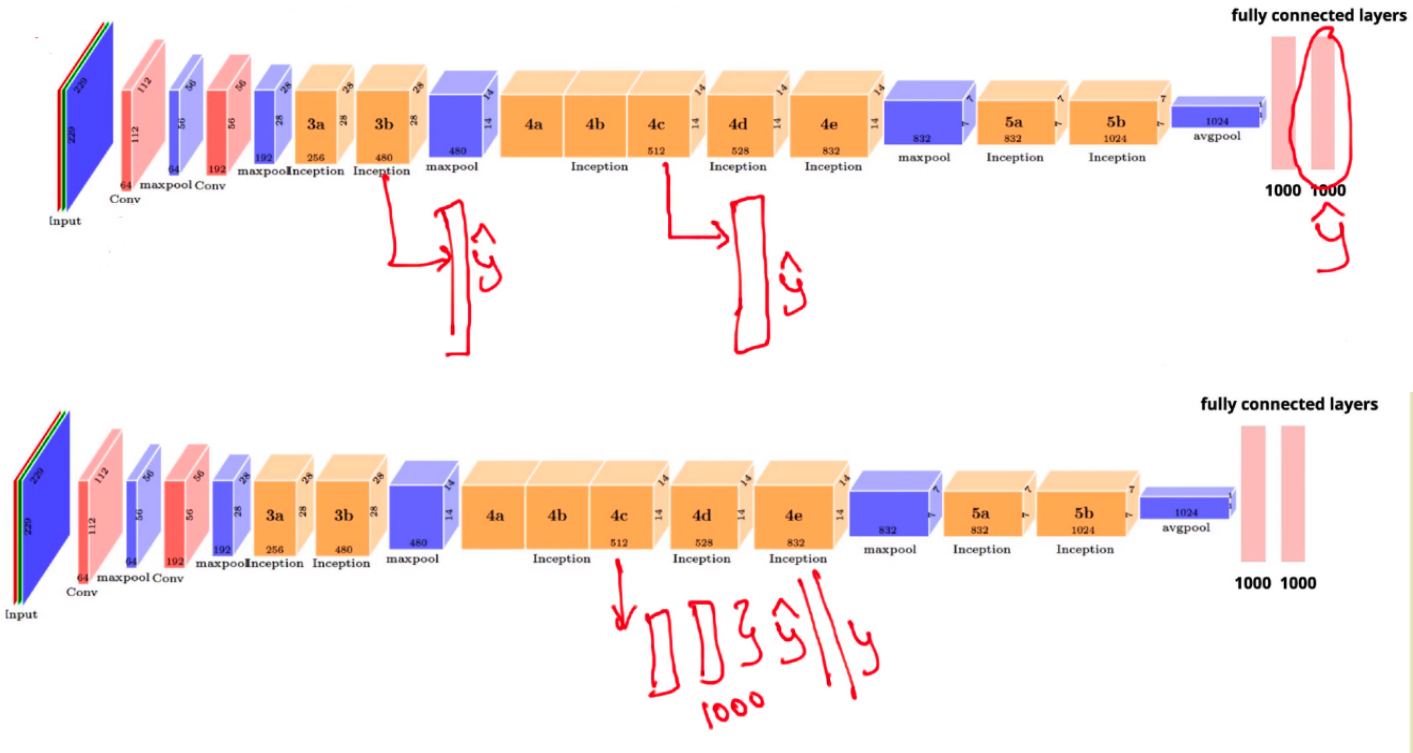
We pass the input through all of the layers, compute an output and based on true output, we compute the loss value using any loss function say Cross-Entropy, we then train this network using backpropagation; what that means is that using this loss value, we are going to update the parameters in all of the 22 layers. Since this is a very deep network, it might be the case that some of the gradients might vanish, and to deal with this we compute the auxiliary loss.

Let say after the layer/block **4c**(below diagram), we get some output and from this output, we try to predict our distribution over **1000** classes that we have, so we try to predict some **dummy y_hat**, and say we also try to predict this **dummy y_hat** at some



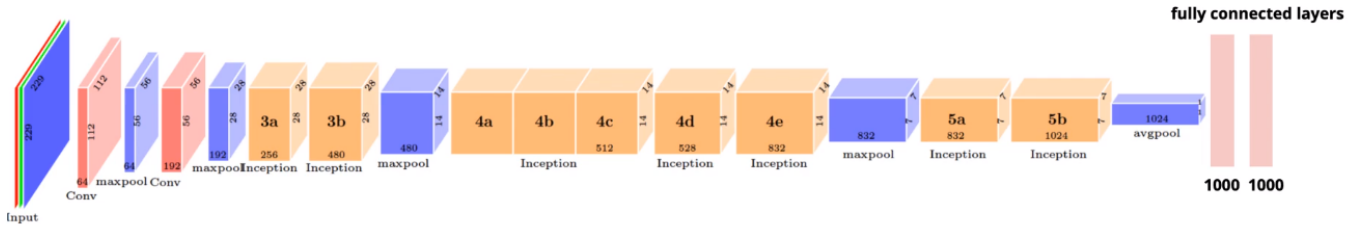
The way to look at this is that we have this final output layer at **5b** which we shrink using Average pooling and we pass that through a fully connected layer and predict the **y_hat**; the same thing we could do at any of the intermediate layers also and we can predict the **y_hat** at those layers as well. So, in a way, we have computed the deep representation of the image using the partial network and from there we try to make a prediction, we do this after a few layers again and then we have the final prediction(at the last layer).

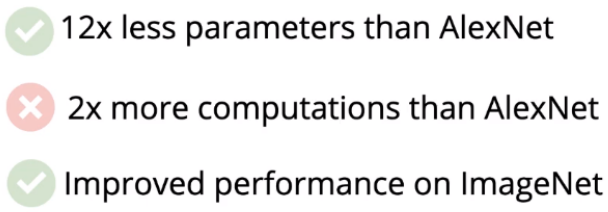
We compute the loss value at all of the representations and backpropagate to the initial layers based on the loss value through these intermediate representations. **This loss at the intermediate step is known as Auxiliary Loss.**



getting vanished are low in this case. Auxiliary loss helps to train the network better.

The net result of using all the tricks in GoogleNet is depicted below:





Deep Learning Computer Vision Artificial Intelligence Machine Learning

Convolutional Network